Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2012) 1–10

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Dynamic Tracing of Parallel I/O Activity in HPC Applications

Seong Jo Kim[a,*], Seung Woo Son[b], Mahmut Kandemir[a], Wei-keng Liao[b], Rajeev Thakur[c], Alok Choudhary[b]

[a]*Department of Computer Science and Engineering*
The Pennsylvania State University, University Park, PA 16802, USA
[b]*Department of Electrical Engineering and Computer Science*
Northwestern University, Evanston, IL 60208, USA
[c]*Mathematics and Computer Science Division*
Argonne National Laboratory, Argonne, IL 60439, USA

Abstract

Most I/O- and data-intensive scientific applications access multiple layers in the parallel I/O software stack during execution. Typical I/O requests from these applications may include accesses to high-level I/O libraries such as Parallel netCDF and HDF5, the MPI I/O library, and parallel file systems. To design and implement parallel applications that exercise such parallel I/O software stack, one must understand the flow of interactions between I/O calls across the entire I/O stack. This would in turn help one describe I/O behavior and thus exploit the potential performance in the different layers of the storage hierarchy. In this paper, we propose a Pin-based dynamic instrumentation framework to understand the complex interactions of I/O from the applications through multiple I/O libraries to the underlying parallel file systems without any modification of the code. We also present the overheads incurred by the proposed dynamic instrumentation tool. When our tested application is executed using a process count of 32, 64, 128, and 256, the overheads we observed are 38.7%, 66%, 68.9%, and 78.4%, respectively.

Keywords: Dynamic instrumentation, MPICH2, PVFS, Parallel netCDF, HDF5, I/O software stack, Pin

1. Introduction

Frequently, users of high-performance computing (HPC) systems face an interesting situation: it is not the CPU, memory, or network that limits the performance of their applications; it is the storage system. That is, I/O behavior is the primary factor that determines the overall application performance. Therefore, understanding how the parallel I/O system operates and the issues involved is critically important when tuning an application to meet the requirements for a particular system or deciding an I/O solution to match expected workloads.

Unfortunately, understanding I/O behavior is not trivial since it is a result of complex interactions between the hardware and a number of software layers, collectively referred to as the *I/O software stack*. Figure 1 illustrates a typical I/O stack for an HPC system. At the lowest level is the storage hardware. This layer consists of the disks, controllers, and interconnection network across multiple physical devices. At this level, data is usually accessed at

[*]Corresponding author
Email addresses: seokim@cse.psu.edu (Seong Jo Kim), sson@eecs.northwestern.edu (Seung Woo Son), kandemir@cse.psu.edu (Mahmut Kandemir), wkliao@ece.northwestern.edu (Wei-keng Liao), thakur@mcs.anl.gov (Rajeev Thakur), choudhar@eecs.northwestern.edu (Alok Choudhary)

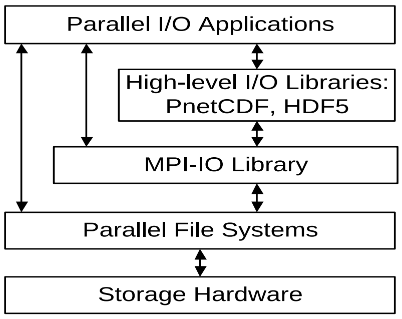

Parallel I/O Applications
High-level I/O Libraries: PnetCDF, HDF5
MPI-IO Library
Parallel File Systems
Storage Hardware

Figure 1: Parallel I/O software stack

the granularity of blocks, either physical disk blocks or logical blocks across multiple physical devices such as in a RAID array. Above the storage hardware are the parallel file systems, such as PVFS [1], GPFS [2], Lustre [3], and PanFS [4]. The roles of parallel file system are to manage the data on the storage hardware, to present this data as a directory hierarchy, and to coordinate accesses to files and directories in a consistent manner. The MPI-IO [5] library sits on top of the parallel file systems. The MPI-IO interface is the standard parallel I/O interface and exists on most high-performance parallel computing platforms today. It provides the API and optimizations such as data caching, and process coordination [6, 7, 8, 9, 10, 11]. While the MPI-IO interface is effective and advantageous because of its performance and portability, most scientific applications work with structured data. For this reason many scientific applications take advantage of a high-level API written on top of MPI-IO (e.g., Parallel netCDF [12] or HDF5 [13]). These high-level interfaces allow application programmers to better describe how their applications access shared storage resources. Further, they provide data abstractions that match the way the scientific applications view data.

One approach to understanding I/O behavior is to let the application programmers or scientists manually instrument the I/O software stack. Unfortunately, this approach is extremely difficult and error-prone. In fact, instrumenting even a single I/O call may necessitate modifications to numerous files to pass the trace information from the application to multiple I/O software layer below. Worse, a high-level I/O call from the application program can be fragmented into multiple calls (subcalls) in the MPI-IO library. Since most parallel scientific applications today are expected to run on large-scale systems with hundreds of thousands of processors in order to achieve better performance, even collecting and analyzing trace log data from them are laborious and burdensome.

Motivated by these observations, we have developed a *dynamic* performance analysis and visualization tool for parallel I/O. Instead of manually instrumenting applications and other components of the parallel I/O stack components, we leverage a lightweight binary instrumentation using Pin [14] to implement our current prototype of the tool. In other words, our tool performs the instrumentation in the binary code of the MPI-IO library and the underlying parallel file system, PVFS, at runtime. Therefore, our tool provides the language-independent instrumentation working with scientific applications written in C/C++ or Fortran. Further, our tool requires neither source code modification nor recompilation of the applications and parallel I/O stack components. Lastly, a unique aspect of our implementation is that it provides a hierarchical view for the parallel I/O. In our implementation, each MPI I/O call has a unique identification number in the MPI-IO layer and is passed to the underlying file systems with trace information. This mechanism helps associate the MPI I/O call from the applications with its subcalls in the PVFS layer in a systematic way. In addition, our tool provides detailed I/O performance metrics—including I/O latency at each I/O software stack layer, disk throughput, and the number of I/O calls from the PVFS client to the PVFS server—for each MPI I/O call. To our knowledge, no currently existing tools provide this functionality.

While our tool can be used for dynamic (runtime) I/O optimizations, our main goal in this paper is to present its implementation details and quantify its overhead. The rest of this paper is organized as follows. Related work is discussed in Section 2. Section 3 presents the technical details of our code instrumentation and latency computation. An experimental evaluation of the tool is presented in Section 4, followed by our concluding remarks in Section 5.

2. Related Work

Over the past decade many code instrumentation tools have been developed and tested that target different machines and applications. ATOM [15] inserts probe code into the program at compile time. Dynamic code instrumentation, on the other hand, intercepts the execution of an executable at runtime to insert user-defined codes at different points of interest. HP's Dynamo [16] monitors an executable's behavior through interpretation and dynamically selects hot instruction traces from the running program. DynamoRIO [17] is a binary package with an interface for both dynamic instrumentation and optimization. PIN [14] is designed to provide a functionality simulator to the ATOM toolkit; but unlike ATOM which instruments an executable statically by rewriting it, PIN inserts the instrumentation code dynamically while the executable is executing. In comparison, Daikon [18] uses instrumentation to extract program invariants.

Several techniques have been proposed in the literature to reduce instrumentation overheads. Dyninst and Paradyn use fast breakpoints to reduce the overheads incurred during instrumentation. Both are designed for dynamic instrumentation [19]. In comparison, FIT [20] is a static system that aims at retargetability rather than instrumentation optimization. INS-OP [21] is also a dynamic instrumentation tool that applies transformations to reduce the overheads in the instrumentation code. In [22], Vijayakumar et al. propose an I/O tracing approach that combines aggressive trace compression; however, their strategy does not provide flexibility in terms of target metric specification.

Tools such as CHARISMA [23], Pablo [24], and Tuning and Analysis Utilities (TAU) [25] collect and analyze file system traces [26]. Paraver [27] is designed to analyze MPI, OpenMP, Java, hardware counters profile, and operating system activity. Open | SpeedShop [28] is targeted to support performance analysis of applications. Kojak [29] aims at the development of a generic automatic performance analysis environment for parallel programs. Darshan [30] captures I/O behavior such as file access patterns in applications, and Vampir [31] provides an analysis framework for MPI applications. Stack Trace Analysis Tool (STAT) [32] is designed to help debug large-scale parallel programs. It gathers and merges multiple stack traces across space, one from each of a parallel application's processes, and across time through periodic samples from each process. HPCToolkit [33] also uses sampling for measurement and analysis of program performance.

For the MPI-based parallel applications, several tools have been developed, such as MPI Parallel Environment (MPE) [34] and mpiP [35]. The latter is a lightweight profiling tool for identifying communication operations that do not scale well in the MPI-based applications. It reduces the amount of profile data and overheads by collecting only statistical information on MPI functions. Typically, the trace data generated by these profiling tools are visualized using tools such as Jumpshot [36], Nupshot [37], Upshot [38], and PerfExplorer [39]. PIOViz [40] supports the combined tracing of MPI client processes and PVFS server processes by source instrumentation; the traces of PIOViz are shown by Jumpshot. Static code instrumentation is also supported in [41] to trace parallel I/O from the MPI library to PVFS servers.

Our work differs from these efforts primarily because we provide a dynamic instrumentation framework to entirely trace parallel I/O from the MPI library to the underlying parallel file system. Unlike static code instrumentation, our implementation inserts instrumentation probe code at runtime and generates trace information to analyze the performance of I/O. Since our tool performs the dynamic instrumentation in the *binary code* of the MPI library and PVFS, it does *not* need any source code modification and recompilation of the application, the high-level scientific libraries such as PnetCDF and HDF5, the MPI library, and PVFS. Consequently, our approach provides portability, manageability, and flexibility to understand and analyze parallel I/O in HPC systems. Moreover, we support various analytical functionalities and metrics such as latency, throughput, and call information to investigate detailed I/O behavior.

3. Technical Details

We provide here details about code instrumentation, latency, and throughput.

3.1. Our Dynamic Instrumentation

The main goal behind this is to provide a *dynamic* instrumentation framework for scientific applications with minimal impact on the performance. Our current implementation uses Pin [14], a lightweight binary instrumentation

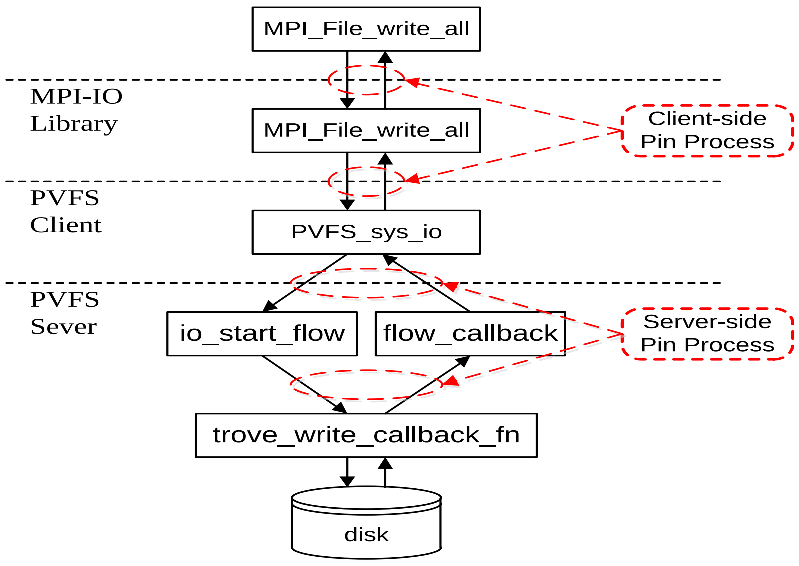


Figure 2: Overview of our dynamic instrumentation framework. The Pin process on the client side creates trace log files at the boundary of the MPI library and PVFS client. The Pin on the server side, on the other hand, produces trace log files for the PVFS server and disk.

tool to instrument binary code of the MPI library and PVFS. As a result, our tool does not require source code modification and recompilation of parallel I/O software stack components.

Figure 2 illustrates the overview of our Pin-based framework. This figure is intended to explain the flow of MPI I/O call and how the framework carries out the dynamic instrumentation when a MPI_File_write_all() function is issued. In the figure, two Pin profiling processes on the client side and the server side generate trace log files at the border of each layer—the MPI library, PVFS client, PVFS server, and disk. The log file contains trace information of each layer such as rank, mpi_call_id, pvfs_call_id, pvfs_server_id, I/O type, and timestamp of each layer's boundary.

Figure 3 shows how the trace information is passed to the PVFS server from the application. When the MPI_File_write_all() function is called, the Pin process on the client side generates trace information such as rank, mpi_call_id, and pvfs_call_id for the function. In the figure, MPI_File_write_all() calls a PVFS_sys_write() function. The PVFS_sys_write() function is replaced with PVFS_sys_io() by definition in the MPI library. The last argument in PVFS_sys_io() is PVFS_HINT_NULL (a NULL value of a PVFS_hints structure). The Pin process packs the trace information including rank, mpi_call_id, and pvfs_call_id for the MPI I/O operation into a new PVFS_hints structure and replaces PVFS_HINT_NULL with PVFS_hints. At the starting point of the client, the Pin process produces a log file using the information. The PVFS_hints structure passed from the MPI layer is packed into a state machine control block (smcb) in the PVFS client and is passed to the PVFS server. Note that a high-level MPI I/O call can be fragmented into multiple calls when the requested size of the I/O call to be written or read is bigger than that of the buffer in the MPI library, which is 16 MB by default. In this situation, pvfs_call_id can be used to differentiate the split calls for one mpi_call_id from others.

At the starting point of the server operation, io_start_flow(), the Pin process receives the address that points to the flow_descriptor structure. From the flow_descriptor, it locates the PVFS_hints list. After that, it searches the PVFS_hints structure generated by the Pin process in the client from the hint list and extracts the trace information passed from it. Using this information, it then creates a server log file. In the io_start_io() function, the PVFS_hints structure is packed into a flow_descriptor structure that contains all information about the request I/O.

At the point of disk operation, trove_write_callback_fn(), the Pin process acquires the address that points to the flow_descriptor from the first argument (void *user_ptr) in the function. It then finds the PVFS_hints from the flow_descriptor and produce a log file for disk operation using the information from the hint. Note that since the flow_descriptor moves in the entire PVFS server, the trace information can be easily accessible at any point in the

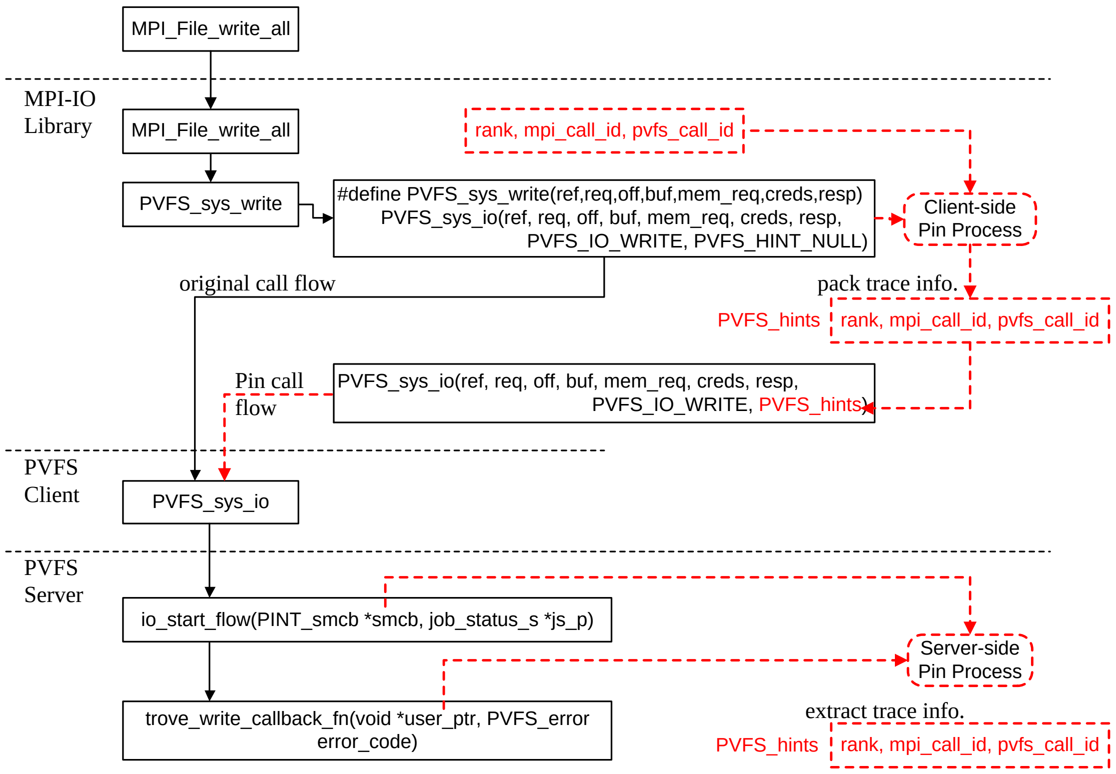

MPI_File_write_all
MPI-IO Library
MPI_File_write_all
PVFS_sys_write
rank, mpi_call_id, pvfs_call_id
#define PVFS_sys_write(ref,req,off,buf,mem_req,creds,resp) PVFS_sys_io(ref, req, off, buf, mem_req, creds, resp, PVFS_IO_WRITE, PVFS_HINT_NULL)
Client-side Pin Process
original call flow
pack trace info.
PVFS_hints
rank, mpi_call_id, pvfs_call_id
Pin call flow
PVFS_sys_io(ref, req, off, buf, mem_req, creds, resp, PVFS_IO_WRITE, PVFS_hints)
PVFS Client
PVFS_sys_io
PVFS Server
io_start_flow(PINT_smcb *smcb, job_status_s *js_p)
Server-side Pin Process
trove_write_callback_fn(void *user_ptr, PVFS_error error_code)
extract trace info.
PVFS_hints
rank, mpi_call_id, pvfs_call_id

Figure 3: Detailed description demonstrating how the Pin process passes the trace information. The Pin process creates new PVFS_hints with rank, mpi_call_id, and pvfs_call_id. It then replaces PVFS_HINT_NULL in PVFS_sys_io with PVFS_hints containing the trace information.

server without any complexity.

3.2. Computation Methodology

To help users understand and analyze I/O behavior for the scientific applications, our tool provides latency and throughput statistics. Figure 4 illustrates the computation of latency and throughput. The value of I/O latency computed at each layer is the *maximum* of the I/O latencies from the layers below it:

$$Latency_i = Max(Latency_{i-1}A, Latency_{i-1}B, Latency_{i-1}C). \tag{1}$$

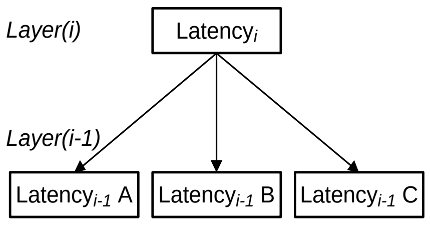
However, the computation of I/O throughput in Figure 4b is additive; in other words, the I/O throughput computed at any layer is the *sum* of the I/O throughput from the layers below it:

$$Throughput_i = \sum(Thpt_{i-1}A, Thpt_{i-1}B, Thpt_{i-1}C). \tag{2}$$

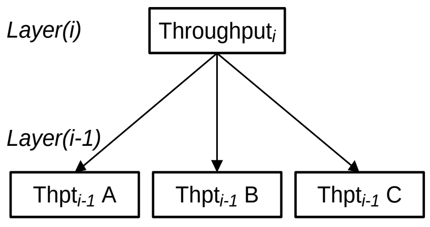
Figure 5 demonstrates how the latency is computed at each layer in more detail. At each layer our tool generates a unique ID such as *process_id*, *mpi_call_id*, *pvfs_call_id*, and *server_id* when an I/O call is passed. This unique number (ID) is cumulatively carried down to the sublayers. All information for the I/O call passed through the entire I/O stack is stored in the last layer. By matching and identifying these IDs, we can easily relate the high-level MPI I/O call to its fragmented subcalls below.

When a collective I/O call[1] is issued, the size of the requested I/O operation might be bigger than that of the buffer size in the MPI library, which is 16 MB by default. In this case the I/O call can be fragmented into multiple subcalls.

[1]Although each process may need to access several noncontiguous portions of a file, the requests of different processes are interleaved and may together span large contiguous portions of the file to improve I/O performance.



(a) Computation of I/O latency



(b) Computation of I/O throughput

Figure 4: Computation of latency and throughput. I/O latency computed at each layer is equal to the maximum value of the I/O latencies obtained from the layers below it. In contrast, I/O throughput is the sum of I/O throughput coming from the layers below.

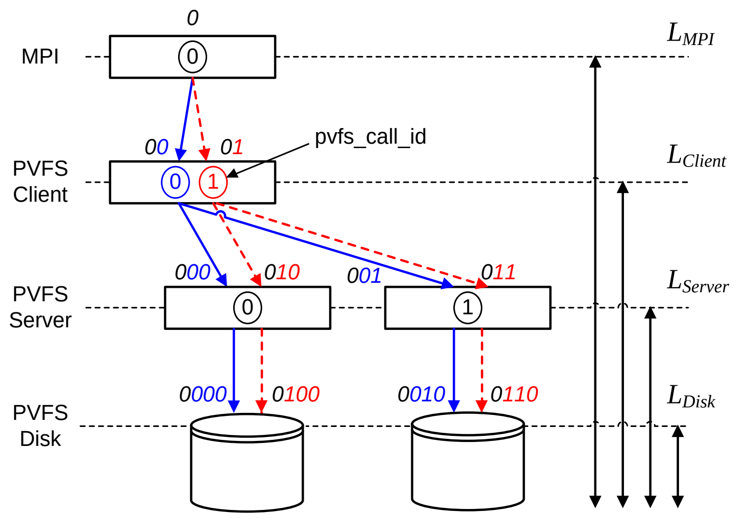


Figure 5: Computation of latency. mpi_call_id *0* is fragmented into two pvfs_call_id's, *0* and *1*. When each split I/O call, *00* and *01* for the mpi_call_id 0 reach servers 0 and 1, the cumulative trace information is *000* and *001* for cumulative ID 00 (solid blue arrow), and *010* and *011* for ID 01 (dotted red arrow).

For example, in the figure mpi_call_id *0* is fragmented into two pvfs_call_id's *0* and *1*. In the PVFS client layer, each split I/O call has its own ID, *00* and *01* for the mpi_call_id 0, respectively. When these calls reach servers 0 and 1, the cumulative trace information is *000* and *001* for cumulative ID 00 (solid blue arrow)and *010* and *011* for ID 01 (dotted red arrow). This relationship is maintained until the end of the I/O stack is reached. Therefore, for mpi_call_id 0, the latency computed at the PVFS client layer is

$$Latency_{client} = \sum(L_{00}, L_{01}), \tag{3}$$

and the latency at the PVFS server layer is

$$Latency_{server} = \sum(Max(L_{000}, L_{001}), Max(L_{010}, L_{011})). \tag{4}$$

4. Evaluation

Our framework for the scientific applications is evaluated on the Breadboard [42] cluster at Argonne National Laboratory (ANL). Each node of this cluster consists 8 quad-core Intel Xeon Processors and 16 GB main memory.

# of processes	Parallel I/O stack		overhead
	uninstrumented	instrumented	
32	128.09	179.09	38.7%
64	64.31	106.77	66%
128	50.44	85.80	68.9%
256	50.77	90.62	78.4%

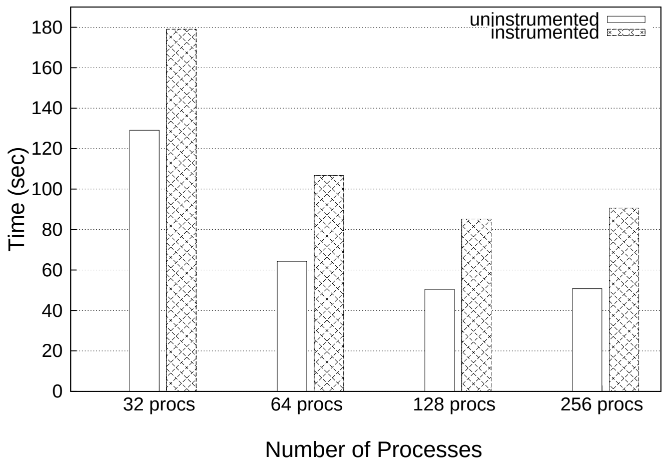


Figure 6: Execution time of S3D I/O depending on the number of processes. Comparing with the uninstrumented I/O software stack, the overhead caused by dynamic instrumentation with 32, 64, 128, and 256 processes is 38.7%, 66%, 68.9%, and 78.4%, respectively.

Therefore, each physical node can support 32 MPI processes. We evaluated our implementation running on 1 metadata server, 8 I/O servers, and 256 processes. In our evaluation, we use pvfs-2.8.2, mpich2-1.4, and pnetcdf-1.2.0 as our parallel I/O software stack.

To demonstrate the effectiveness of the framework, we ran an I/O-intensive benchmark, S3D-IO [43]. S3D I/O is the I/O kernel of S3D application, a parallel turbulent combustion application using a direct numerical simulation solver developed at Sandia National Laboratories (SNL). S3D solves fully compressible Navier-Stokes, total energy, species, and mass continuity equations coupled with detailed chemistry. A checkpoint is performed at regular intervals; its data consists primarily of the solved variables in 8-byte, three-dimensional arrays. This checkpoint data can be used to obtain several more derived physical quantities of interest. Therefore, most of the checkpoint data is maintained for later use. At each checkpoint, four global arrays—representing the variables of mass, velocity, pressure, and temperature—are written to files. All four arrays share the same size for the lowest three spatial dimensions X, Y, and Z and are partitioned among the MPI processes along with X-Y-Z dimensions. S3D I/O supports MPI-IO, PnetCDF, and HDF5 interfaces.

In our evaluation, we maintain the block size of the partitioned X-Y-Z dimension as 200×200×200 in each process. With the PnetCDF interface, it produces three checkpoint files, 976.6MB each. Figure 6 compares the execution time of S3D I/O when running on un-instrumented I/O stack and dynamically instrumented I/O stack. We observe that, with the process counts of 32, 64, 128, and 256, the overheads incurred by our proposed dynamic instrumentation are 38.7%, 66%, and 68.9%, and 78.4%, respectively.

Plotted in Figure 7 is the latency spent in the MPI library, PVFS client, and PVFS server from the perspective of the rank 0 with 256 processes. It can be observed that a large fraction of the time spent in the server is due to disk operations. In S3D I/O, three checkpoint files are produced by 12 collective I/O calls, and each checkpoint file is generated by 0~3, 4~7, and 8~11, respectively. For example, the first checkpoint file is opened by the mpi_call_id 0. The four arrays of mass, velocity, pressure, and temperature are sequentially written by the mpi_call_id 0, 1, 2, and 3. We observe from Figure [**?**] the latency difference between the MPI library and the PVFS client. In S3D I/O, all

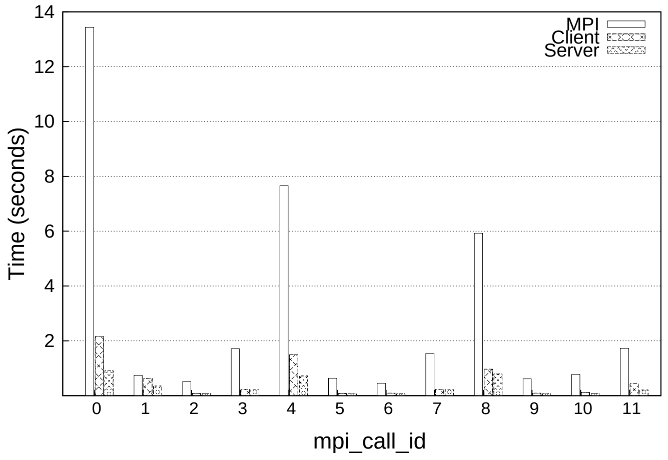


Figure 7: Execution time of S3D I/O for each mpi_call_id. For mpi_call_id the latency spent in the MPI library, PVFS client, and PVFS server is plotted in order. Most of the time spent in the server layer is for disk operations. The latency difference between the MPI library and client is due to optimization and synchronization.

the joined processes heavily exchanges data for optimization such as data sieving [10] and two-phase I/O [11]. The optimization and synchronization result in the overhead in the MPI library. We also notice that the latency in the MPI library for mpi_call_id 0, 4, and 8 is longer than that of the others. These calls first open the checkpoint file and write the mass array, which is the largest one among the four arrays. Because of the overhead to open the checkpoint file and the size of the mass array, these calls spend longer time in the MPI library. In this experiment, the mpi_call_id 0, 4, and 8 are fragmented into 6 subcalls to open the checkpoint file and write the mass array, respectively. The mpi_call_id 3, 7, and 11 for the temperature is split into 2 subcalls each (see Figure 8.)

Figure 8 plots the I/O throughput of S3D I/O from mpi_call_id 0 to 3. The x-axis is a pair of (*mpi_call_id* - *pvfs_call_id*). In the figure mpi_call_id 0 are fragmented into 6 subcalls, (*0-0*) ∼ (*0-5*). We observe that the bandwidth of the first call (0-0) is relatively low because it takes longer time to open the checkpoint file.

S3D I/O uses the collective I/O during the operation. In the collective I/O operation, all the joined processes heavily exchange data to optimize the requests. In addition, communication and synchronization among the processes mainly cause the overhead in the MPI library. Based on this understanding, scientists and application programmers can customize the existing code to reduce the overhead, specifically file-open operation. Also, performance engineers may improve the performance in the MPI library.

5. Conclusions

Understanding I/O behavior is one of the most important factors for efficient execution of scientific applications. The first step in understanding I/O behavior is to instrument the flow of I/O call. Unfortunately, performing manual code instrumentation is extremely difficult and error-prone since the characteristics of I/O are a result of complex interactions of both hardware and multiple layers of software components. Because of the scale of the current HPC systems, collecting and analyzing trace information to understand I/O characteristics are challenging and daunting tasks. To alleviate these difficulties, we propose a framework to instrument I/O stack dynamically. Instead of manual instrumentation, our software framework performs the dynamic instrumentation in the binary code of the MPI library and PVFS. The tool inserts trace information into a PVFS_hints structure and passes it into the sublayers at runtime. This innovative method can provide a hierarchical view of the I/O call from the MPI library to the PVFS server without source code modification or recompilation of the given applications, the high-level libraries, the MPI library, and the parallel file system.

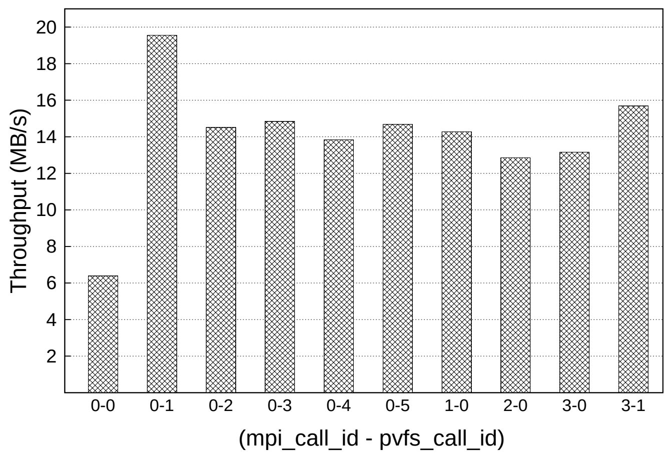


Figure 8: I/O throughput of S3D I/O. The graph plots the throughput for the corresponding subcalls from 0 to 3. The four arrays—mass, velocity, pressure, and temperature—are written by mpi_call_id 0, 1, 2, and 3, in order. The first call (0-0) opens the checkpoint file. Depending on the size of the request I/O, the I/O call is fragmented into multiple subcalls. The I/O call for the mass, 0, is fragmented into 6 subcalls, and the I/O call for the temperature, 3, is split into 2 subcalls.

We used a scientific application benchmark, S3D I/O, to evaluate our proposed framework. Changing the number of processes to run S3D I/O, the overhead induced by our implementation is about 63% on average. Our tool provides several metrics to understand and analyze I/O behavior, latency of each layer, and disk throughput for the I/O. The results from these metrics contribute to evaluating and tuning the applications and I/O software stack. Work is underway (i) to minimize the overheads incurred by our scheme under large process counts, and (ii) to employ our framework for runtime (dynamic) I/O optimizations.

Acknowledgments

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] P. Carns, W. Ligon III, R. Ross, R. Thakur, PVFS: A parallel file system for Linux clusters, in: Proceedings of the 4th annual Linux Showcase & Conference-Volume 4, USENIX Association, 2000, pp. 28–28.

[2] F. Schmuck, R. Haskin, GPFS: A shared-disk file system for large computing clusters, in: Proceedings of the First USENIX Conference on File and Storage Technologies, Citeseer, 2002, pp. 231–244.

[3] P. Schwan, Lustre: Building a file system for 1000-node clusters, in: Proceedings of the 2003 Linux Symposium, Citeseer, 2003.

[4] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, B. Zhou, Scalable performance of the panasas parallel file system, in: Proceedings of the 6th USENIX Conference on File and Storage Technologies, USENIX Association, 2008, p. 2.

[5] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir, MPI - The Complete Reference: Volume 2, The MPI-2 Extensions (1998).

[6] A. Ching, A. Choudhary, K. Coloma, W. Liao, R. Ross, W. Gropp, Noncontiguous I/O accesses through MPI-IO.

[7] X. Ma, M. Winslett, J. Lee, S. Yu, Improving MPI-IO output performance with active buffering plus threads.

[8] K. Coloma, A. Choudhary, W. Liao, L. Ward, E. Russell, N. Pundit, Scalable high-level caching for parallel I/O.

[9] W. Liao, A. Ching, K. Coloma, A. Choudhary, et al., An implementation and evaluation of client-side file caching for MPI-IO, in: 2007 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2007, p. 49.

[10] R. Thakur, W. Gropp, E. Lusk, Data sieving and collective I/O in ROMIO, in: Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation, 1998, pp. 182–189.

[11] J. Del Rosario, R. Bordawekar, A. Choudhary, Improved parallel I/O via a two-phase run-time access strategy, ACM SIGARCH Computer Architecture News 21 (5) (1993) 31–38.

[12] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, M. Zingale, Parallel netCDF: A high-performance scientific I/O interface, in: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, IEEE Computer Society, 2003, p. 39.

[13] HDF5: Hierarchical Data Format, `http://www.hdfgroup.org/HDF5/`.

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: ACM SIGPLAN Notices, Vol. 40, ACM, 2005, pp. 190–200.

[15] A. Srivastava, A. Eustace, ATOM: A system for building customized program analysis tools, in: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, ACM, 1994, pp. 196–205.

[16] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: a transparent dynamic optimization system, in: ACM SIGPLAN Notices, Vol. 35, ACM, 2000, pp. 1–12.

[17] D. Bruening, Efficient, transparent, and comprehensive runtime code manipulation, Ph.D. thesis, Citeseer (2004).

[18] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, C. Xiao, The Daikon system for dynamic detection of likely invariants, Science of Computer Programming 69 (1-3) (2007) 35–45.

[19] J. Hollingsworth, O. Niam, B. Miller, Z. Xu, M. Gonçalves, L. Zheng, MDL: A language and compiler for dynamic program instrumentation, in: pact, Published by the IEEE Computer Society, 1997, p. 201.

[20] B. De Bus, D. Chanet, B. De Sutter, L. Van Put, K. De Bosschere, The design and implementation of FIT: a flexible instrumentation toolkit, in: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, ACM, 2004, pp. 29–34.

[21] N. Kumar, B. Childers, M. Soffa, Low overhead program monitoring and profiling, in: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, ACM, 2005, pp. 28–34.

[22] K. Vijayakumar, F. Mueller, X. Ma, P. Roth, Scalable I/O tracing and analysis, in: Proceedings of the 4th Annual Workshop on Petascale Data Storage, ACM, 2009, pp. 26–31.

[23] N. Nieuwejaar, D. Kotz, A. Purakayastha, S. Ellis, M. Best, File-access characteristics of parallel scientific workloads, Parallel and Distributed Systems, IEEE Transactions on 7 (10) (1996) 1075–1089.

[24] H. Simitci, Pablo MPI Instrumentation User's Guide.

[25] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, B. Mohr, A scalable approach to MPI application performance analysis, Recent Advances in Parallel Virtual Machine and Message Passing Interface (2005) 309–316.

[26] S. Browne, J. Dongarra, K. London, Review of performance analysis tools for MPI Parallel Programs, NHSE Review 3 (1).

[27] V. Pillet, J. Labarta, T. Cortes, S. Girona, Paraver: A tool to visualize and analyze parallel code, in: Transputer and occam developments: WoTUG-18: proceedings of the 187th world occam and Transputer User Group Technical Meeting, 9th-13th April 1995, Manchester, UK, Vol. 44, Ios Pr Inc, 1995, p. 17.

[28] Open—SpeedShop, URL: http://www.openspeedshop.org/wp/.

[29] B. Mohr, F. Wolf, Kojak–a tool set for automatic performance analysis of parallel programs, Euro-Par 2003 Parallel Processing (2004) 1301–1304.

[30] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, K. Riley, 24/7 characterization of petascale i/o workloads, in: Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on, IEEE, 2009, pp. 1–10.

[31] W. Nagel, A. Arnold, M. Weber, W. Nagel, A. Arnold, M. Weber, K. Solchenbach, Vampir: Visualization and analysis of mpi resources.

[32] D. Arnold, D. Ahn, B. De Supinski, G. Lee, B. Miller, M. Schulz, Stack trace analysis for large scale debugging, in: 2007 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2007, p. 64.

[33] J. Mellor-Crummey, Hpctoolkit: Multi-platform tools for profile-based performance analysis, in: 5th International Workshop on Automatic Performance Analysis (APART), Phoenix, AZ, November, 2003.

[34] A. Chan, W. Gropp, E. Lusk, Users guide for mpe extensions for mpi programs (2003).

[35] J. Vetter, C. Chambreau, mpiP: Lightweight, Scalable MPI Profiling, URL: http://www.llnl.gov/CASC/mpiP.

[36] O. Zaki, E. Lusk, W. Gropp, D. Swider, Toward scalable performance visualization with Jumpshot, International Journal of High Performance Computing Applications 13 (3) (1999) 277–288.

[37] E. Karrels, E. Lusk, Performance analysis of MPI programs, in: Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing, SIAM Publications, 1994, pp. 195–200.

[38] V. Herrarte, E. Lusk, Studying parallel program behavior with Upshot, Tech. Rep. ANL-91/15, Argonne National Laboratory, Argonne, IL (1991).

[39] K. Huck, A. Malony, PerfExplorer: A performance data mining framework for large-scale parallel computing.

[40] T. Ludwig, S. Krempel, M. Kuhn, J. Kunkel, C. Lohse, Analysis of the mpi-io optimization levels with the pioviz jumpshot enhancement, Recent Advances in Parallel Virtual Machine and Message Passing Interface (2007) 213–222.

[41] S. Kim, Y. Zhang, S. Son, R. Prabhakar, M. Kandemir, C. Patrick, W. Liao, A. Choudhary, Automated tracing of i/o stack, Recent Advances in the Message Passing Interface (2010) 72–81.

[42] `http://wiki.mcs.anl.gov/radix/index.php/Breadboard`.

[43] R. Sankaran, E. Hawkes, J. Chen, T. Lu, C. Law, Direct numerical simulations of turbulent lean premixed combustion, in: Journal of Physics: conference series, Vol. 46, IOP Publishing, 2006, p. 38.